Methodology

Setup: Built and ran `dist/server.js` with the V8 CPU profiler in IntelliJ (or VS Code), capturing a 15-second profile under load.
Load: Issued Postman scripts (100 randomized POSTs across `/address/distance`, `/address/city`, `/address/request`, `/address/count`).
Memory: Took a heap snapshot mid-run and analyzed retained objects.

Key Findings

Metric	Value/Function
Overall CPU Utilization	~13% of core capacity under load
Top CPU Consumers	• `writeToStandardOut` (console.log): ~5.7% • `Socket._write` (net.js): ~4% • `cityLookup` (address.service.js): 2–3 % • `flush` (address.service.js): 2-3%
Memory Footprint	~8.4 MB heap at snapshot
Heap Composition	Standard arrays, strings, objects; no leaks detected

Problems Identified

1. **Excessive synchronous logging** (`writeToStandardOut`):
   - Consumes ~5.7% of the CPU on every request, blocking the event loop.
2. **I/O-bound hot spots** (`Socket._write`, downstream lookups):
   - The network writes account consumes ~4% of the CPU; the city lookup logic consumes another 2–3%.
3. **Lack of high-resolution timing**:
   - Millisecond-level per-method durations weren't captured; profiling shows only relative load.

Future Changes

- **Optimize logging:**
  - Switch to an asynchronous, leveled logger (e.g., Pino or Winston with `async` transport).
  - Disable or throttle debug and info logs on high-volume endpoints.

- **Batch or cache lookups:**

    - Implement in-memory LRU caching for repeated city and zip code resolutions.
    - Parallelize any sequential downstream calls in `cityLookup` and `flush`.
- **Stream large payloads:**
    - For large JSON responses, use streaming APIs (`res.write()` + `JSONStream`) to reduce buffer overhead.
- **Enhance observability:**
    - Add middleware to log per-request timings (e.g., using `response-time` or a custom timer).
    - Correlate high-latency requests with specific endpoints in logs.
- **Prepare for scale:**
    - Use Node's `cluster` module or a process manager (PM2/Docker replicas) to utilize all CPU cores.
    - Monitor network I/O as request volume grows; consider back-pressure strategies.

Expected Impact

- **Lower CPU overhead** on each request (< 3 % average), freeing cycles for business logic.
- **Improved throughput and responsiveness** under concurrent load.
- **Maintain stable memory usage**, with headroom for additional features.